Invited paper to be presented at the Fifth IEEE International Symposium on High Assurance Systems Engineering (HASE), November, 2000, Albuquerque NM.

Disappearing Formal Methods*

Abstract

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

In his book *The Invisible Computer* [11], Donald Norman describes how most technological innovations begin as very complex products that require highly skilled operators, and become widely accepted and generally useful only as the products manage more of their own functions and present a simple interface to their operators. Early radios, for example, had dozens of controls to adjust such arcane features as regeneration and filter bandwidth, and operators were expected to understand "the fundamentals of radio reception." A modern radio, on the other hand, is trivial to operate.

Most current formal methods are like those early radios: they are difficult to use and fit awkwardly into traditional software engineering processes. For example, formal methods generally require a formal specification or model that is quite different to the other documents and descriptions produced during design and whose construction is an additional burden for the developers. Formal analysis requires mastery of the recondite skills needed to guide theorem provers and model checkers—skills that that are more concerned with the formal methods technology employed than with the design being analyzed.

There are those who believe that the benefits of formal specification outweigh the cost of their introduction and that the best way forward is to develop a workforce for whom formal specifications are the primary means of documentation and communication—rather in the way that mathematical notation and analysis underlie every other engineering discipline. This is certainly a worthwhile goal, and may be necessary for critical systems that require the highest levels of assurance, but the path to reach that goal from current practice is difficult, and it is hard to persuade practitioners to undertake it if there are few intermediate rewards along the way.

So the topic I want to consider is how to make formal methods invisible in Norman's sense: how to make them "disappear" into the familiar fabric of software engineering practice. The point is not to "dumb down" formal methods, but to take some of its technologies and make them sufficiently convenient, powerful, and useful that practitioners adopt them willingly—and ask for more. In this way they may be persuaded to begin the journey that leads to a more formal and scientific basis for software engineering.

It has to be admitted that abstract "correctness" is simply not a concern for most software products: they have merely to be "good enough." Thus, more effective debugging and testing are more potent inducements to adopt formal methods than proofs of correctness or systematic refinement. Among formal methods, model checking is generally used for debugging, and has achieved a modest degree of acceptance in some industries for that reason. However, model checking still requires construction of special "downscaled" finite-state models. Theorem proving is less widely accepted because it yields less useful information on failure, and is much less automated. It is impossible to fully automate theorem proving without the risk that the automation will fail to prove some true theorems. Fortunately, it is possible to combine theorem proving and model checking in ways that reduce their individual weaknesses and produce a combination that is significantly better than either alone. This approach is based on automated abstraction: theorem proving is used to construct the "downscaled" models needed for model checking [13]. Automated theorem proving can be used because the construction is tolerant of the failure to prove some true theorems (the abstraction will be less precise than otherwise, but may still be adequate for its purpose). I describe this approach, and how it can be extended from debugging to verification, in a little more detail elsewhere [12]. Prototype tools are being developed to support this approach in which it is intended that the formal machinery should "disappear" into familiar environments such as those for simulation. For example, in SAL [2], the simulation model is translated into the SAL intermediate language, the formal analysis is conducted in a largely automated manner, and the results are then translated back. The user experience should be similar to using

*This work was supported by the Darpa MoBIES program through contract F33615-00-C-1700 from the Air Force Wright Laboratory.

exhaustive simulation to explore an aspect of a design. Related approaches apply model checking to abstracted real programs [1, 8].

Other ways in which formal methods can "disappear" inside familiar tools in ways that augment their power include extended static checking [4], in which theorem proving is used greatly to increase the power of typechecking, consistency checking for class descriptions [9], reachability analysis (e.g., in a Statechart, the user could ask "show me a trace that leads to this state and that one simultaneously" [3]), and test case generation [10] and analysis [6].

To support these kinds of "invisible" formal methods, I expect that many of the capabilities of monolithic formal methods tools will be "liberated" and be made available as standalone components that can be used inside other tools. Candidate components include integrated decision procedures for powerful combinations of theories, symbolic simulators, model checkers, constraint solvers, computer algebra systems, abstractors and concretizers, slicers, and counterexample analyzers.

As the power of these components becomes available in many familiar tools, I look forward to the "disappearance" of formal methods—and their greatly increased use.

Acknowledgements

The ideas outlined here have evolved through discussions with my colleagues Judy Crow, Ursula Martin, Sam Owre, Hassen Saïdi, and Natarajan Shankar.

References

[1] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In Havelund et al. [7], pages 113–130.

[2] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Muñoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. M. Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, June 2000. NASA Langley Research Center. Proceedings available at `http://shemesh.larc.nasa.gov/fm/Lfm2000/Proc/`.

[3] T. Bienmüller, W. Damm, and H. Wittke. The STATEMATE verificaiton environment: Making it real. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification, CAV '2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 561–567, Chicago, IL, July 2000. Springer-Verlag.

[4] D. L. Detlefs. An overview of the Extended Static Checking system. In *First Workshop on Formal Methods in Software Practice (FMSP '96)*, pages 1–9, San Diego, CA, Jan. 1996. Association for Computing Machinery.

[5] N. Halbwachs and D. Peled, editors. *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.

[6] K. Havelund. Using runtime analysis to guide model checking of Java programs. In Havelund et al. [7], pages 245–264.

[7] K. Havelund, J. Penix, and W. Visser, editors. *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, Stanford, CA, Aug.–Sept. 2000. Springer-Verlag.

[8] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[9] D. Jackson. Enforcing design constraints with object logic. In J. Palsberg, editor, *Seventh International Static Analysis Symposium (SAS'00)*, volume 1824 of *Lecture Notes in Computer Science*, pages 1–21, Santa Barbara CA, June 2000. Springer-Verlag.

[10] T. Jéron and P. Morel. Test generation derived from model-checking. In Halbwachs and Peled [5], pages 108–121.

[11] D. A. Norman. *The Invisible Computer: Why Good Products Can Fail, the Personal Computer Is So Complex, and Information Appliances Are the Solution*. The MIT Press, 1999.

[12] J. Rushby. From refutation to verification. In *Formal Description Techniques and Protocol Specification, Testing and Verification FORTE XIII/PSTV XX 2000*, Pisa, Italy, Oct. 2000. Chapman & Hall, UK. To appear.

[13] H. Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [5], pages 443–454.